# Finalized Security Assessment

# LOCKON Finance

Mar 27, 2024

This smart contract audit report was created by Bunzz Audit. It utilized a database of over 1000 contract vulnerability patterns, comparing the project's contract against this database with AI to conduct a comprehensive diagnosis of vulnerabilities

BUNZZ AUDIT

## Table of Contents

## Summary

The smart contract audit revealed several security vulnerabilities that could potentially be exploited by malicious actors. These issues range from medium to low severity and include problems such as unfair reward distribution, unsafe token approval practices, and lacks certain best practices that could lead to unforeseen behavior. This contract also has some room to improve gas efficiencies. It is imperative that these vulnerabilities are addressed to ensure the security of the smart contract before deployment.

Source URLs: https://gitlab.com/lockon-finance/lock-contracts/-/tree/5577152c164aeca8c8247a18d433d972efdedf04

## Coverage

In this audit, we checked a total of 72 types of vulnerability patterns.

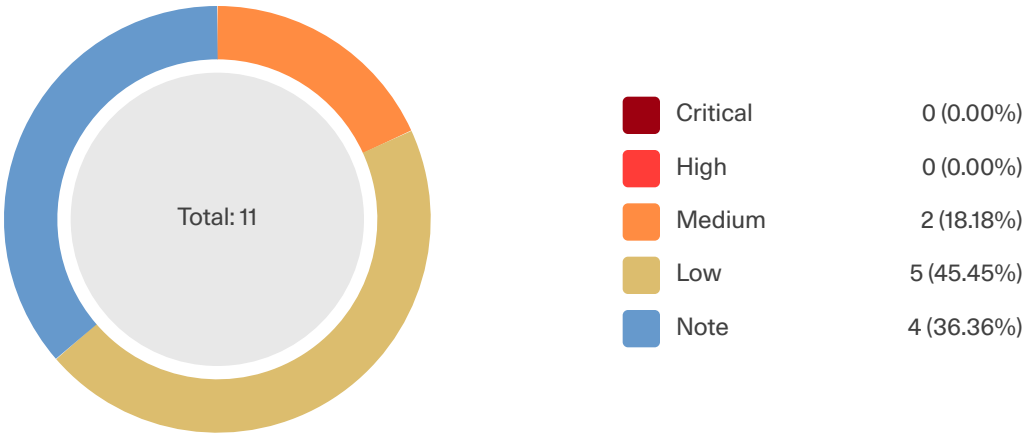
Found 9 (12.50%)
Not Found 63 (87.50%)

| Title | Found |
|---|---|
| Front Running | - |
| Back Running | - |
| Sandwiching | - |
| Transaction order dependency | - |
| Fake tokens | - |
| Fake contracts | - |
| On-chain oracle manipulation | - |
| Governance attack | - |
| Token standard incompatibility | - |
| Flash liquidity borrow, purchase, mint, or deposit | - |
| Unsafe call to phantom function | - |
| One DeFi protocol dependency | - |
| Unfair slippage protection | - |
| Unfair liquidity providing | - |
| Unsafe or infinite token approval | ID-6 |
| Delegatecall injection | - |
| Unhandled or mishandled exception | - |
| Locked or frozen tokens | - |
| Absence of code logic or sanity check | ID-1, ID-2 |
| Casting | - |
| Unbounded operation, including gas limit and call-stack depth | ID-3 |
| Arithmetic mistakes | ID-7 |
| Inconsistent access control | - |
| Visibility errors, including unrestricted action | - |
| Direct call to untrusted contract | - |
| Function Default Visibility | - |
| Integer Overflow and Underflow | - |
| Outdated Compiler Version | - |

| | |
|---|---|
| Floating Pragma | - |
| Unchecked Call Return Value | - |
| Unprotected Ether Withdrawal | - |
| Wrong Comparison Operator | - |
| UINT256 could be more gas efficient than smaller types | - |
| Magic numbers should be replaced with constants | ID-8 |
| Unprotected SELFDESTRUCT Instruction | - |
| Reentrancy | - |
| State Variable Default Visibility | - |
| Uninitialized Storage Pointer | - |
| Assert Violation | - |
| Use of Deprecated Solidity Functions | - |
| Delegatecall to Untrusted Callee | - |
| DoS with Failed Call | - |
| Transaction Order Dependence | - |
| Authorization through tx.origin | - |
| Block values as a proxy for time | ID-4, ID-5 |
| Signature Malleability | - |
| Incorrect Constructor Name | - |
| Shadowing State Variables | - |
| Weak Sources of Randomness from Chain Attributes | - |
| Missing Protection against Signature Replay Attacks | - |
| Lack of Proper Signature Verification | - |
| Requirement Violation | - |
| Write to Arbitrary Storage Location | - |
| Incorrect Inheritance Order | - |
| Insufficient Gas Griefing | - |
| Arbitrary Jump with Function Type Variable | - |
| DoS With Block Gas Limit | - |
| Typographical Error | - |
| Right-To-Left-Override control character (U+202E) | - |
| Presence of unused variables | - |
| Unexpected Ether balance | - |
| Hash Collisions With Multiple Variable Length Arguments | - |
| Message call with hardcoded gas amount | - |
| Code With No Effects | - |
| Unencrypted Private Data On-Chain | - |
| Constant variables should be marked as private | ID-9 |
| Reading array length in for-loops | - |
| Checked arithmetic for for-loops | - |
| ++i costs less gas than i++ | ID-10 |
| IERC20.transfer does not support all ERC20 tokens | ID-11 |

| | |
|---|---|
| Unrestricted minting | - |
| Trust Issue Of Admin Roles | - |

## Findings




| ID | Title | Severity | Status |
|---|---|---|---|
| ID-1 | Zero address check | Medium | Fixed |
| ID-2 | Update lastRewardTimestamp without condition | Medium | Fixed |
| ID-3 | Unbound loop iteration | Low | Fixed |
| ID-4 | Block values as a proxy for time | Low | Acknowledged |
| ID-5 | Block values as a proxy for time | Low | Acknowledged |
| ID-6 | Unsafe token approval | Low | Fixed |
| ID-7 | Insufficient reward supply check | Low | Fixed |
| ID-8 | Magic numbers should be replaced with constants | Note | Fixed |
| ID-9 | Visibility of constant variables | Note | Fixed |
| ID-10 | ++i costs less gas than i++ | Note | Fixed |
| ID-11 | IERC20.transfer does not support all ERC20 tokens | Note | Fixed |

Details

## ID-1: Zero address check

### Severity

Medium

### Location

LockToken.sol:LockToken:initialize

### Description

The initialize function does not check for zero addresses for ownerAddress and managementAddress, which could lead to minting tokens to an inaccessible address.

### How to fix

Add require statements to check for zero addresses before minting tokens.

### Code

```
function initialize(
        string memory name,
        string memory symbol,
        address ownerAddress,
        address managementAddress
    ) external initializer {
        __ERC20_init_unchained(name, symbol);
        __Ownable_init_unchained(ownerAddress);
        __UUPSUpgradeable_init();
        uint256 amountMintToOwner = (MAX_SUPPLY * 4000) / BASE_DENOMINATOR;
        uint256 amountMintToManagement = (MAX_SUPPLY * 6000) / BASE_DENOMINATOR;
        _mint(ownerAddress, amountMintToOwner * (10 ** uint256(decimals())));
        _mint(
            managementAddress,
            amountMintToManagement * (10 ** uint256(decimals()))
        );
    }
```

### Code Suggestion

```
require(ownerAddress != address(0), 'LockToken: ownerAddress is the zero address');
require(managementAddress != address(0), 'LockToken: managementAddress is the zero
address');
```

### Status

Fixed

### Comment

The issue has been fixed.

## ID-2: Update lastRewardTimestamp without condition

### Severity

Medium

### Location

IndexStaking.sol:IndexStaking:updatePool

### Description

The function 'updatePool' updates the pool's lastRewardTimestamp to the current block timestamp without considering if the update is necessary, which could lead to incorrect reward calculations if called multiple times within the same block.

### How to fix

Add a check to ensure that the pool's rewards have changed before updating the lastRewardTimestamp.

### Code

```
function updatePool(address _stakeToken) public {...}
```

### Code Suggestion

```
if (lockReward > 0) { poolInfo.lastRewardTimestamp = block.timestamp; }
```

### Status

Fixed

### Comment

The issue has been fixed.

## ID-3: Unbound loop iteration

### Severity

Low

### Location

contracts/Airdrop.sol:Airdrop:distributeAirdropReward

### Description

The 'distributeAirdropReward' function contains an unbounded loop that iterates over potentially large arrays without a bound on their size, which could lead to hitting the block gas limit and causing transaction failure.

### How to fix

Implement a limit on the size of the arrays that can be processed in a single transaction or use a pull over push strategy for reward distribution to prevent excessive gas consumption.

Code

```
function distributeAirdropReward(
        address[] calldata _listUserAddress,
        uint256[] calldata _amounts
    ) external whenNotPaused onlyDistributeGrantedOrOwner {
        uint256 listUserLen = _listUserAddress.length;
        require(
            _amounts.length == listUserLen,
            "Airdrop: The list for user address and amount value must have equal
length"
        );
        uint256 totalAmount;
        for (uint256 i; i < listUserLen; ) {
            require(
                _listUserAddress[i] != address(0),
                "Airdrop: Zero address is not allowed"
            );
            require(_amounts[i] != 0, "Airdrop: Zero amount is not allowed");
            userPendingReward[_listUserAddress[i]] += _amounts[i];
            totalAmount += _amounts[i];
            unchecked {
                i++;
            }
        }
        totalPendingAirdropAmount += totalAmount;
        emit AirdropRewardDistributed(
            msg.sender,
            listUserLen,
            totalPendingAirdropAmount
        );
    }
```

Code Suggestion

```
require(listUserLen <= MAX_DISTRIBUTION_COUNT, "Airdrop: Too many addresses");

// Define a constant for the maximum number of distributions allowed
uint256 public constant MAX_DISTRIBUTION_COUNT = 100; // Example value
```

Status

Fixed

### Comment

The issue has been fixed.

## ID-4: Block values as a proxy for time

### Severity

Low

### Location

MerkleAirdrop.sol:MerkleAirdrop:claimPendingReward, LockStaking:LockStaking:updatePool, IndexStaking.sol:IndexStaking:deposit, Airdrop.sol:Airdrop:claimPendingReward

### Description

The use of `block.timestamp` to determine the start of an airdrop can be manipulated by miners, potentially leading to incorrect assumptions about the timing of events.

### How to fix

Use an external time source or oracle, or implement a tolerance interval to mitigate the risk of miner manipulation.

### Code

```
require(block.timestamp >= startTimestamp, "Airdrop: Airdrop not start");
```

### Code Suggestion

```
Consider adding a tolerance interval for the timestamp check or using an external
oracle to provide a reliable timestamp. For example:

function withinTolerance(uint256 _timestamp) private view returns (bool) {
    uint256 tolerance = 180; // 3 minutes tolerance
    return (_timestamp <= block.timestamp + tolerance) && (_timestamp >=
block.timestamp - tolerance);
}

require(withinTolerance(startTimestamp), "Airdrop: Airdrop not start");
```

### Status

Acknowledged

### Comment

The project team is aware of this issue and decided not to fix it because it does not require the exact timing.

## ID-5: Block values as a proxy for time

### Severity

Low

### Location

LockStaking:LockStaking:basicRate, LockonVesting.sol:LockonVesting:currentTotalClaimable, LockonVesting.sol:LockonVesting:_claimable, LockonVesting.sol:LockonVesting:deposit, LockonVesting.sol:LockonVesting:getVestingEndTime

### Description

The use of `block.timestamp` for calculating claimable amounts can be manipulated or imprecise, leading to incorrect calculations and potential security risks.

### How to fix

No change is needed unless the exact timing is critical. If it is, then an external time source should be considered. Replace the use of `block.timestamp` with a reliable external time source or oracle, or design the logic to be resilient to the imprecision and potential manipulation of `block.timestamp`.

### Code

```
function basicRate() public view returns (uint256) {
        uint256 secondsPassed = (block.timestamp - lockTokenReleasedTimestamp);
        return PRECISION + (secondsPassed * basicRateDivider);
    }
```

### Code Suggestion

```
Consider introducing an external time oracle or a mechanism for the contract owner
to update the time, if high precision is required. Additionally, implement a
maximum drift for the `block.timestamp` to ensure that the time used in
calculations does not deviate significantly from the actual time.
```

### Status

Acknowledged

### Comment

The project team is aware of this issue and decided not to fix it because it does not require the exact timing.

## ID-6: Unsafe token approval

### Severity

Low

### Location

LockStaking.sol:LockStaking:claimPendingReward

### Description

The contract does not check for existing allowances before setting a new allowance, which can be exploited by an attacker through a front-running attack.

### How to fix

Implement checks for existing allowances and use the 'increaseAllowance' and 'decreaseAllowance' pattern to safely manage allowances.

### Code

```
function claimPendingReward(
        string calldata _requestId,
        uint256 _commissionSharingReward,
        bytes memory _signature
    ) external whenNotPaused nonReentrant {
        ...
        if (totalCumulativeReward != 0) {
            lockToken.approve(lockonVesting, totalCumulativeReward);
            ILockonVesting(lockonVesting).deposit(
                msg.sender,
                totalCumulativeReward,
                LOCK_STAKING_VESTING_CATEGORY_ID
            );
            _currentUserInfo.cumulativePendingReward = 0;
        }
        ...
    }
```

### Code Suggestion

```
uint256 currentAllowance = lockToken.allowance(address(this), lockonVesting);
if (currentAllowance < totalCumulativeReward) {
    lockToken.safeIncreaseAllowance(lockonVesting, totalCumulativeReward -
currentAllowance);
}
ILockonVesting(lockonVesting).deposit(
    msg.sender,
    totalCumulativeReward,
    LOCK_STAKING_VESTING_CATEGORY_ID
);
```

### Status

Fixed

### Comment

The issue has been fixed.

## ID-7: Insufficient reward supply check

### Severity

Low

### Location

LockStaking:LockStaking:updatePool

### Description

The code may not properly handle the case where 'rewardSupply' is less than 'lockReward', which could lead to insolvency if not managed correctly.

### How to fix

Ensure that the reward distribution logic is atomic and that 'rewardSupply' is always sufficient to cover 'lockReward'.

### Code

```
require(rewardSupply >= lockReward, 'LOCK Staking: Reward distributed exceed
supply');
```

### Code Suggestion

```
require(rewardSupply >= lockReward && rewardSupply >= currentRewardAmount, 'LOCK
Staking: Insufficient reward supply');
```

### Status

Fixed

### Comment

The project team changed the specification and this condition is no longer necessary.

## ID-8: Magic numbers should be replaced with constants

### Severity

Note

### Location

LockToken.sol:LockToken:initialize

### Description

Hardcoded values 4000 and 6000 are used for calculating mint amounts, which reduces code readability and maintainability.

### How to fix

Define the numbers 4000 and 6000 as constants with descriptive names to indicate their purpose.

### Code

```
uint256 amountMintToOwner = (MAX_SUPPLY * 4000) / BASE_DENOMINATOR;
uint256 amountMintToManagement = (MAX_SUPPLY * 6000) / BASE_DENOMINATOR;
```

### Code Suggestion

```
uint256 private constant OWNER_ALLOCATION_PERCENTAGE = 4000;
uint256 private constant MANAGEMENT_ALLOCATION_PERCENTAGE = 6000;

uint256 amountMintToOwner = (MAX_SUPPLY * OWNER_ALLOCATION_PERCENTAGE) /
BASE_DENOMINATOR;
uint256 amountMintToManagement = (MAX_SUPPLY * MANAGEMENT_ALLOCATION_PERCENTAGE) /
BASE_DENOMINATOR;
```

### Status

Fixed

Comment

The issue has been fixed.

## ID-9: Visibility of constant variables

### Severity

Note

### Location

contracts/Airdrop.sol:Airdrop:AIRDROP_VESTING_CATEGORY_ID, contracts/LockToken.sol:LockToken:MAX_SUPPLY, contracts/LockToken.sol:LockToken:BASE_DENOMINATOR, contracts/LockStaking.sol:LockStaking:PRECISION, contracts/LockStaking.sol:LockStaking:LOCK_STAKING_VESTING_CATEGORY_ID, contracts/IndexStaking.sol:IndexStaking:INDEX_STAKING_VESTING_CATEGORY_ID

### Description

These public constant variables could be declared as private to improve gas efficiency.

### How to fix

Change the visibility of the constants variable to private.

### Code

```
uint256 public constant AIRDROP_VESTING_CATEGORY_ID = 2;
```

```
uint256 public constant MAX_SUPPLY = 10_000_000_000;
```

```
uint256 public constant BASE_DENOMINATOR = 10_000;
```

```
uint256 public constant PRECISION = 1e12;
```

```
uint256 public constant LOCK_STAKING_VESTING_CATEGORY_ID = 0;
```

```
uint256 public constant INDEX_STAKING_VESTING_CATEGORY_ID = 1;
```

### Code Suggestion

```
uint256 private constant AIRDROP_VESTING_CATEGORY_ID = 2;
```

### Status

Fixed

### Comment

The issue has been fixed.

## ID-10: ++i costs less gas than i++

### Severity

Note

### Location

contracts/IndexStaking.sol:IndexStaking:initialize, contracts/IndexStaking.sol:IndexStaking:updateCurrentRewardAmount, contracts/Airdrop.sol:Airdrop:distributeAirdropReward

### Description

The use of post-increment i++ in a for loop is less gas-efficient than pre-increment ++i because it requires an additional temporary variable to hold the original value of i before incrementing.

### How to fix

Replace the post-increment i++ with a pre-increment ++i to save gas.

### Code

```
for (uint256 i; i < len; ) {
    ...
    unchecked {
        i++;
    }
}
```

```
for (uint256 i; i < currentNumOfPools; ) {
    ...
    unchecked {
        i++;
    }
}
```

```
for (uint256 i; i < listUserLen; ) {
    require(
        _listUserAddress[i] != address(0),
        "Airdrop: Zero address is not allowed"
    );
    require(_amounts[i] != 0, "Airdrop: Zero amount is not allowed");
    userPendingReward[_listUserAddress[i]] += _amounts[i];
    totalAmount += _amounts[i];
    unchecked {
        i++;
    }
}
```

### Code Suggestion

```
for (uint256 i; i < len; ) {
    ...
    unchecked {
        ++i;
    }
}
```

### Status

Fixed

### Comment

The issue has been fixed.

## ID-11: IERC20.transfer does not support all ERC20 tokens

### Severity

Note

### Location

Airdrop.sol:Airdrop:claimPendingReward

### Description

While the code uses the approve function, it does not explicitly use the safeApprove function from the SafeERC20 library. However, since the SafeERC20 library is being used, it is likely that the library handles the approve function correctly.

### How to fix

Ensure that the SafeERC20 library's safeApprove function is used if the standard approve function does not handle non-standard tokens correctly.

### Code

```
lockToken.approve(lockonVesting, userAmount);
```

### Code Suggestion

```
lockToken.safeApprove(lockonVesting, userAmount);
```

### Status

Fixed

### Comment

The issue has been fixed.

## Disclaimer

We conducted our review of the smart contract codes solely based on the materials and documentation provided by the project under audit (the "Project").

Our audit employed a fine-tuned Artificial Intelligence (AI) system, which incorporates (i) a database of known vulnerability patterns that we have collected up until the date of our review for this audit, and (ii) results from a selection of existing contract analysis tools available as of the date of our review for this audit. While we endeavor to ensure the highest possible quality and accuracy of the results produced by our fine-tuned AI, we must clarify that the results are based on the state of the AI technology and understanding of smart contracts as of the date mentioned, and consequently absolute completeness and infallibility of the AI-generated results cannot be guaranteed.

In addition to the aforementioned AI-driven analysis, we may conduct manual audits. These are grounded in the knowledge and expertise we have accumulated up to the date of our review for this audit. The purpose of these manual audits is to identify and assess vulnerabilities specific to the project under audit.

Blockchain and smart contract technologies continue to evolve and may be susceptible to unforeseen risks and flaws. Consequently, while it is possible to minimize smart contract security risks, their complete elimination is inherently unattainable. Therefore, our audit does not claim to provide an exhaustive or all-encompassing review of all potential vulnerabilities.

Lastly, it is important to clarify that the scope of our audit is strictly limited to the analysis of smart contracts. Our audit does not extend to other layers or components, including but not limited to hardware, operating systems, programming languages, compilers, protocols, platforms, virtual machines, and imported libraries.

**DISCLAIMER**: You agree to the terms set forth in this disclaimer by reading this report or any part of it. Bunzz Pte. Ltd. and its affiliates (including shareholders, subsidiaries, employees, directors, and other representatives if any) (“Bunzz”) provide this report for information purposes only. No party, including third parties, has the right to rely on this report or its contents. Bunzz assumes no responsibility or duty of care to any person and makes no warranty or representation about the accuracy or completeness of this report to any person. Bunzz provides this report "as-is," without any conditions, warranties, or other terms of any kind, and hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, fitness for purpose, merchantability, or non-infringement). Except to the extent that it is prohibited by law, Bunzz excludes all liability and responsibility, and you or any other person will not have any claim against Bunzz for any amount or kind of loss or damage that may result to you or any other person (including, without limitation, any direct, indirect, special, punitive, consequential, or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, whether in tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report, its use, the inability to use it, the results of its use, and any reliance on this report. For the avoidance of doubt, no one should consider or rely upon this report, its content, access, or usage as any form of financial, investment, tax, legal, regulatory, or other advice.